Chapter 11. JAX-RPC and JAXM

With most software technologies, standardization occurs when the technology has gained a certain amount of momentum. Well, SOAP and web services have gained the requisite momentum, and standards (and their resulting acronyms) are sprouting up. There are a slew of API standards evolving for manipulating XML in Java, all of which fall under the umbrella of the Java APIs for XML (JAX). JAX is not a product — it's an API definition, and as such, it's no different from many of the countless API specifications associated with Java. However, since XML is such a wide-ranging area, JAX is made up of many components. Two of these are of particular interest to us: JAX-RPC and JAXM. JAX-RPC is the Java API for XML-based RPC. This is the API that, over time, I'd expect most significant SOAP RPC implementations to follow. JAXM is the Java API for XML Messaging, and, like JAX-RPC, it will likely become widely accepted.

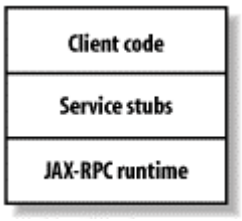
These two APIs are new, and are not yet ready for the production world. The specifications for JAXM and JAX-RPC are in public review, and therefore are expected to change before they're complete; however, at this stage, the changes should be relatively minor. Nonetheless, Sun has produced a reference implementation that allows you to start working with these technologies. This reference implementation is included in the Java Web Services Developer Pack (Java WSDP) available at http://java.sun.com/webservices/webservicespack.html. You can treat this release as an early beta; as long as you don't use it for a production server and don't expect it to have all the bugs worked out or the features you like, it will help you get up to speed so that you can build a production server as soon as possible when the standards are finalized. The Web Services Developer Pack uses a Tomcat 4 server, so if you've been using Apache SOAP with Tomcat up to this point, you should be pretty comfortable working with JAX. Even if you haven't been using Tomcat, the installation instructions for the pack are straightforward, and you shouldn't have much trouble with it.

To explore JAX-RPC, we'll build a few examples using the reference implementation of JAX. These examples will be new versions of the examples we developed earlier in this book. We'll move quickly, with the assumption that you've read every chapter up to this point. We won't attempt to cover JAX-RPC and JAXM completely; instead, we'll use this opportunity to take a quick look at how these APIs are structured, and how they compare to the Apache SOAP 2.X and GLUE APIs that we explored in earlier chapters. In other words, we'll be rebuilding a few of the service and client examples from previous chapters using JAX.

11.1 JAX-RPC

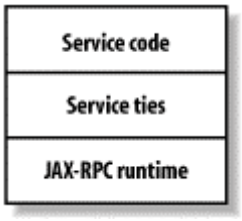
Let's start with JAX-RPC, since we've spent most of our time in the RPC world. Like GLUE, JAX-RPC hides the details of the underlying protocol (SOAP) from the programmer. To hide the underlying protocol, JAX-RPC makes use of objects known as stubs and ties. A *stub* is used by a client application to access a remote service. The stub looks like the service interface, but it runs as part of the local client process. The stub in turn uses the underlying framework to generate an appropriate SOAP message and send it over HTTP. So from the perspective of a client application programmer, if you have the stub you don't need to deal with any of the details of the SOAP message. Figure 11-1 shows the relationship between client code, stubs, and the runtime component of the JAX-RPC client framework.

Figure 11-1. Stubs in the JAX-RPC architecture



A *tie* is conceptually similar to a stub, but exists on the server side. A tie acts as a bridge between the JAX-RPC runtime and the service implementation, and decouples your service code from the JAX-RPC framework. Inbound service messages are handled by the JAX-RPC runtime and passed to the service via a tie object. Service responses move back through the tie to the JAX-RPC runtime, where they are converted to SOAP response or SOAP fault messages and returned over HTTP. Figure 11-2 shows the relationship between service code, service ties, and the JAX-RPC runtime.

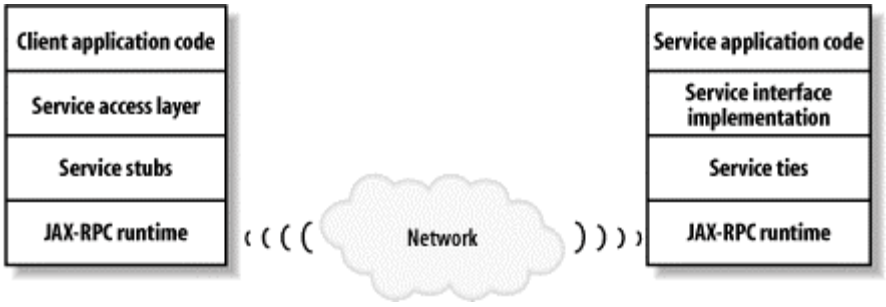
Figure 11-2. Ties in the JAX-RPC architecture



Think for a moment about the RPC mechanisms in Apache SOAP 2.X and GLUE. Apache exposes a great deal of the SOAP machinery to the Java programmer, and asks the programmer to deal with SOAP at a concrete level. Architecturally, JAX-RPC looks quite a bit more like GLUE in that it hides many of the low-level details. This is a positive trend in the evolution of SOAP APIs, whether or not they are proprietary.

Regardless of the complexity of your API or the kind of distributed mechanism you choose, it's good practice to abstract and isolate the part of your software that deals with the distributed computing mechanism. This allows you to swap mechanisms with minimal impact on the rest of your work. The examples in this chapter use this approach, even though the examples are so simple that it may seem like overkill. Figure 11-3 shows the overall architecture of the RPC examples we'll develop using JAX-RPC.

Figure 11-3. RPC Architecture using JAX-RPC



11.2 Working Without Ant

The Java Web Services Developer Pack tutorial relies heavily on a package called Ant. All Java source file compilation, directory creation, JAR file creation, etc., are managed by Ant configuration files. Although Ant is quite popular, and using it to build the kind of code in this chapter is a good idea, we're not going to cover its use here. (For more information about Ant, see *Ant: The Definitive Guide* by Jesse Tilly and Eric Burke.) Instead, I'm going to take you through the manual steps for the examples. If you know how to use Ant, you'll be able to gloss over some of the details in this chapter.

The most critical thing to set up is the classpath. The Early Access Release makes use of 10 JAR files that need to be on your classpath. These files are located in the *common\lib* directory underneath the root of your Developer Pack installation, and it's up to you to make sure they are on your classpath using whatever technique you prefer. Here are the files:

jaxrpc-ri.jar
jaxrpc-api.jar
activation.jar
dom4j.jar
jaxm-api.jar
jaxm-client.jar
log4j.jar
mail.jar
xalan.jar
xerces.jar

11.3 Creating a JAX-RPC Service

Back in Chapter 9 we talked about building proxy services, which sit between client applications and other services. In that chapter, the back-end service with which the proxies communicated was located at *http://mindstrm.com:8004/glue/urn:CorpDataServices*. Let's do the same thing again, only this time we'll build our proxy service using the JAX-RPC reference implementation.[1] This example allows us to see how JAX-RPC interoperates with GLUE, and gives us a chance to build both a service and a client using JAX-RPC.

The first step in building the service is to write a Java interface for it, which I'll call `javasoap.book.ch11.services.IStockServiceProxy`. This interface must implement the `java.rmi.Remote` interface because JAX-RPC makes use of the Java RMI package. This strategy is fairly common because Java RMI provides a good distributed computing abstraction. The methods in the interface should be defined to throw `RemoteException`, which also comes from the `java.rmi` package. We'll define one method, `getStockQuote( )`, which takes a string parameter for the stock symbol and returns an instance of `javasoap.book.ch9.services.ProxyQuote`. There's no need to create a new class for returning a quote; we've already done it in Chapter 9. Here's what the code looks like for the `IStockServiceProxy` interface:

[1] I'm assuming you've downloaded and installed the Java Web Services Developer Pack already, and verified that it is working properly.

```
package javasoap.book.ch11.services;

import java.rmi.Remote;
import java.rmi.RemoteException;
import javasoap.book.ch9.services.ProxyQuote;

public interface IStockServiceProxy extends Remote {
    public ProxyQuote getStockQuote(String symbol) throws RemoteException;
}
```

Next, we need to write a Java class that implements the `IStockServiceProxy` interface, which I'll call `javasoap.book.ch11.services.StockServiceProxyImpl`. I don't normally use the `Impl` suffix in my class names, but it's a common practice, and it suits my purposes here. (Ordinarily, I'd just drop the "I" from the interface name and call the implementation class `StockServiceProxy`, but I want to use that name for something else. I want to decouple the class that really performs the work of the service from the classes that work with JAX-RPC or any other API.) So the `StockServiceProxyImpl` class is part of the Service Interface Implementation layer from Figure 11-3. We'll use another class, called `javasoap.book.ch11.services.StockServiceProxy`, to do the actual work. Unlike `StockServiceProxyImpl`, this class doesn't know anything about the JAX-RPC mechanism, nor does it know anything about the `IStockServiceProxy` interface used by the tie object. This allows us to use the class, without modification, regardless of the distributed computing mechanism we're using. In other words, `StockServiceProxy` is part of the Service Application Code layer shown in Figure 11-3. We'll use an instance of `StockServiceProxy` inside the `getStockQuote( )` method of the `StockServiceProxyImpl` class, as shown below:

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;

public class StockServiceProxyImpl implements IStockServiceProxy {

    public ProxyQuote getStockQuote(String symbol) {
        StockServiceProxy sp = new StockServiceProxy( );
        return sp.getStockQuote(symbol);
    }
}
```

Now we'll implement the `StockServiceProxy` class, whose job it is to invoke the `GetQuote()` method on the `urn:CorpDataServices` service. However, I want to hold off on communicating with this GLUE-based service, as that involves the use of its WSDL file. We'll get to that later. But for now, the `getStockQuote( )` method of `StockServiceProxy` will just return an instance of `ProxyQuote` with dummy data. Here's the code:

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;

public class StockServiceProxy {

        public ProxyQuote getStockQuote(String symbol) {

        ProxyQuote pq = new ProxyQuote( );
        pq.setStockSymbol(symbol);
```

```
        // set dummy data for now
        pq.setLast((float)0.0);
        pq.setDiff((float)0.0);
        pq.setTime("NA");
        pq.setVol(0);

        return pq;
    }
}
```

These are all the Java classes and interfaces we'll need to implement the service. Go ahead and compile them. Now it's time to generate the stubs and ties for the service. A command-line tool called `xrpcc` generates these classes. `xrpcc` takes a configuration file as input. This file is an XML file that contains information about the classes, namespaces, service name, etc. I've named this file *CorpDataServices.xml*. Let's take a look at its contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <rmi name="CorpDataServicesProxy"
    targetNamespace="http://mindstrm.com/wsdl"
    typeNamespace="http://mindstrm.com/types">
      <service name="CorpDataServices"
           packageName="javasoap.book.ch11.client">
        <interface name="javasoap.book.ch11.services.IStockServiceProxy"
         servantName="javasoap.book.ch11.services.StockServiceProxyImpl"/>
      </service>
  </rmi>
</configuration>
```

Everything lies underneath the `configuration` tag, which uses the `xmlns` attribute to define the global namespace as `http://java.sun.com/jax-rpc-ri/xrpcc-config`. Since we're using an interface based on RMI, the first child of configuration is named `rmi`.[2] The `name` attribute is used to specify the *model* name, which is essentially the name that will be used as a base for the files generated by this process. You can think of it as the name of the service definition. (It is used for just that purpose in the WSDL file that gets generated.) The `targetNamespace` and `typeNamespace` attributes are used to declare the namespaces that will be associated with the target service and the custom types, respectively. Under the `rmi` tag comes the `service` tag, which provides `xrpcc` information about the Java classes and interface used to implement the service. The `name` attribute contains the service name itself, and the `packageName` attribute contains the name of the Java package to use for the generated client-side code. Under `service` comes the `interface` tag, which tells the tool about the classes that implement the service interface. The `name` attribute contains the fully qualified name of the Java interface for the service, and the `servantName` contains the fully qualified name of the Java class that implements that interface.

So let's go ahead and run `xrpcc` for our service. The root of my working classpath is `e:\projects\soap`, so I'll pass that on the command line as follows:

```
xrpcc -both -keep -classpath e:\projects\soap -d gen CorpDataServices.xml
```

[2] `xrpcc` also allows the use of a WSDL document instead of RMI. We'll use that approach later in the chapter.

The `-both` option tells `xrpcc` that we want it to generate the files for both the client and server side. If for some reason you prefer to generate client-side and server-side files separately, you'd use either `-client` or `-server` instead. If you specify `-keep`, as I did, the Java source files for the generated code will not be deleted. (By default, `xrpcc` compiles the generated code and then deletes the source files, since you don't really need them for anything. I used `-keep` so I could take a peek at the code. It's useful to see what classes and methods `xrpcc` is generating, particularly when you're getting started.) The `-classpath` option specifies the classpath for finding the input class files; the `-d` option specifies the directory to use as the root for the generated files. Finally, the last argument on the command line is the name of the configuration file.

Here's a listing of the files generated by `xrpcc`, organized by the subdirectories within `xrpcc`'s working directory. Items in bold are directory names. For the sake of clarity, I didn't list the Java source files that were retained because of the `-keep` option. (In fact, I deleted them manually to ensure that they weren't packaged for deployment.)

- **gen**
  - CorpDataServices.wsdl
  - CorpDataServicesProxy_Config.properties
- **javasoap\book\ch11\client**
  - CorpDataServicesProxy.class
  - CorpDataServicesProxy_SerializerRegistry.class
  - CorpDataServicesProxyImpl.class
- **javasoap\book\ch11\services**
  - GetStockQuote_RequestStruct.class
  - GetStockQuote_RequestStruct_SOAPSerializer.class
  - GetStockQuote_ResponseStruct.class
  - GetStockQuote_ResponseStruct_SOAPBuilder.class
  - GetStockQuote_ResponseStruct_SOAPSerializer.class
  - IStockServiceProxy_Stub.class
  - IStockServiceProxy_Tie.class
- **javasoap\book\ch9\services**
  - ProxyQuote_SOAPSerializer.class

I placed the files into a separate *gen* directory because they are going to be packaged in a JAR file a little later, and I don't want to package my entire `javasoap.book` package path. The first time I ran `xrpcc`, I noticed that the stub and tie classes were generated in the same package. I don't like to work that way. As you've seen throughout, I've kept my client-side and server-side classes in separate packages, but `xrpcc` doesn't give me that option. In addition, the serializer class (`ProxyQuote_SOAPSerializer`) for the `javasoap.book.ch9.services.ProxyQuote` class was put into the `javasoap.book.ch9.services` package. I suppose this makes sense, though it's not where I would want the serializer; but again, I don't have control over this. One could certainly mess around with the package names in the source code that `xrpcc` generates, and recompile by hand, but I won't bother. This is still an early-access release, and things will change. Just keep in mind that you may not be happy with the package structuring that `xrpcc` creates.

Next, we need to create a deployment descriptor. The information contained in the deployment descriptor is used to configure the service application on the web server.

Here's the deployment descriptor for our service, which needs to be called *web.xml*; I put it into the *gen* directory. The text in bold represents code specific to this example.

```
<?xml version="1.0" encoding="UTF-8"?>

 <!DOCTYPE web-app PUBLIC
     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
     "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
   <display-name>CorpDataServices</display-name>
   <description>Corporate Data Services</description>
   <servlet>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <display-name>JAXRPCEndpoint</display-name>
      <description>
         Endpoint for Corporate Data Services Application
      </description>
      <servlet-class>
        com.sun.xml.rpc.server.http.JAXRPCServlet
      </servlet-class>
      <init-param>
         <param-name>configuration.file</param-name>
         <param-value>
          /WEB-INF/CorpDataServicesProxy_Config.properties
        </param-value>
      </init-param>
      <load-on-startup>0</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>JAXRPCEndpoint</servlet-name>
      <url-pattern>/jaxrpc/*</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>60</session-timeout>
   </session-config>
</web-app>
```

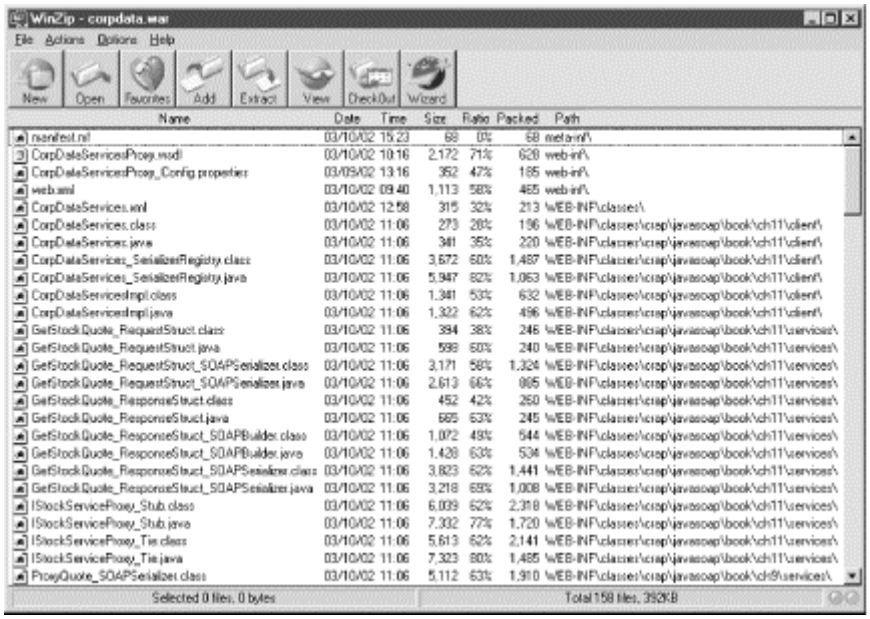
We aren't done yet. Now that we've implemented the Java classes and interfaces for our service, generated ties and stubs, and created a deployment descriptor, the next step is to package the service in a WAR file. A WAR file is a *web application archive*, i.e., a JAR file that contains the files needed to deploy and run a web application according to the Java servlet specification. We'll copy the class files for `IStockServiceProxy`, `StockServiceProxyImpl`, and `StockServiceProxy` into the appropriate directory underneath the *gen* directory, and do the same thing for the `javasoap.book.ch9.services.ProxyQuote` class. We need to organize our files and directories to be a proper WAR file. Everything should start from a root directory that must be called *WEB-INF* (all uppercase letters). Move the *web.xml*, *CorpDataServicesProxy_Config.properties*, and *CorpDataServices.wsdl* files into the *WEB-INF* directory. Now we need to put the Java classes into the proper directories. Create a directory called *classes* underneath the *WEB-INF* directory, and make it the root of the *javasoap* directory. In other words, move the *javasoap* directory and all of its contents into *WEB-INF\classes*. Ant would make managing the process a lot easier: you just set up some configuration files and it does the rest. But now you know what Ant would be doing behind the scenes.

Let's create our WAR file. From the *gen* directory, issue the command:

```
jar cvf corpdata.war WEB-INF
```

This packages everything we need into the WAR file named *corpdata.war*. Figure 11-4 shows the contents of the WAR file using the WinZip program.

Figure 11-4. The contents of corpdata.war



Now are we finished? Not quite. The last step is to deploy the service, which is as simple as copying the WAR file to the *webapps* directory for Tomcat 4 (remember that Tomcat 4 is included with the Java Web Services Developer Pack). So go ahead and do this now. My Web Services Developer Pack is installed on *e:\jwsdp-1_0-ea1*, so I copied *corpdata.war* to *e:\jwsdp-1_0-ea1\webapps*. If you've got Tomcat 4 running, shut it down and restart it; that will get it to read the *corpdata.war* file and deploy the service. You can check to be sure the service is properly deployed by going to *http://localhost:8080/corpdata/jaxrpc* in your browser. If the service is deployed correctly, you'll see the following:

```
A Web Service is installed at this URL.
It supports the following ports: "IStockServiceProxy"
(http://localhost:8080/
corpdata/jaxrpc/IStockServiceProxy)
```

Well, that was easy! Actually, it wasn't as bad as it seems. A lot of the work was in the packaging and deployment. If you use the Ant tool instead of doing it by hand, things go a lot quicker. Remember that this is an early-access release of a reference implementation, so you can be pretty sure that the production systems will have better deployment tools.

Now that we've got a service deployed, let's take a moment to reflect. Is the development process better than the one we used in Apache SOAP? It seems to be. We didn't have to write any code other than that which implemented the service, while in Apache SOAP we had to write quite a bit of code related to the transport, type mapping, etc. In particular, notice that we didn't have to deal with our custom `ProxyQuote` class directly. The stubs and ties handled the entire mapping and serialization process.

How does this compare to GLUE in terms of service development? It's similar, although there are some subtle yet striking differences. As a developer, I know I'd certainly prefer not to deal with any generated code. Why should I have to manage those class files? Why should I have to even know they exist? With GLUE, some of this is handled dynamically, which makes it a bit easier to manage. You still need some client-side stubs, but service development hides everything. In my opinion, that's an advantage.

11.4 Creating a JAX-RPC Client

Before we start writing a client application, let make sure to copy all the class files generated earlier into their proper locations on the local filesystem. Remember that we have them in a WAR file, and we also have them under the *gen\WEB-INF\classes* directory. But neither of these locations is on my local classpath, so now is a good time to put them where they belong so the client application can access them.

Writing a client application for this service is pretty simple. All you need to do is get an instance of the stub that was generated earlier and call the desired service method on that object. Here's the code for a sample application class called `javasoap.book.ch11.client.StockQuoteApp`:

```
package javasoap.book.ch11.client;

import javasoap.book.ch11.services.*;
import javasoap.book.ch9.services.*;

public class StockQuoteApp {
   public static void main(String[] args) {
      try {
         CorpDataServicesProxyImpl proxy =
             new CorpDataServicesProxyImpl(  );
         IStockServiceProxy_Stub stub =
             (IStockServiceProxy_Stub)proxy.getIStockServiceProxy(  );
         stub._setTargetEndpoint(args[0]);
         ProxyQuote quote = stub.getStockQuote("IBM");
         System.out.println("Price is " + quote.getLast(  ));
         System.out.println("Volume is " + quote.getVol(  ));
         System.out.println("Timestamp is " + quote.getTime(  ));
         System.out.println("Diff is " + quote.getDiff(  ));
      } catch (Exception ex) {
        ex.printStackTrace(  );
      }
   }
}
```

We create an instance of `CorpDataServicesProxyImpl`, and then we call its `getIStockServiceProxy( )` method. This gives us a reference to the stub, an instance of `IStockServiceProxy_Stub`. We need to pass the endpoint URL to the stub via its `_setTargetEndpoint( )` method. The URL is *http://localhost:8080/corpdata/jaxrpc/IStockServiceProxy*, which is the location of the service endpoint we deployed earlier. Now we can simply call the `getStockQuote( )` method, which results in an invocation of the service method of the same name. Notice that the return value is an instance of `javasoap.book.ch9.services.ProxyQuote`. Serialization of this custom type is handled automatically on both the client and service side. Running this application results in the following output:

```
Price is 0.0
Volume is 0
Timestamp is NA
Diff is 0.0
```

This is how you might use JAX-RPC to develop both the client and service code for a service. But that won't always be your situation. I mentioned earlier that `xrpcc` can make use of a WSDL file. That might be a nice way to generate stubs for accessing, say, a GLUE service. (And it will probably be the best way to generate stubs for accessing an arbitrary service somewhere on the network.) We'll try that next.

11.5 Generating Stubs from WSDL

Generating client code from WSDL is pretty simple. First we'll modify the *CorpDataServices.xml* file that we used earlier as input to `xrpcc`. We have to remove the entire `rmi` section and replace it with a `wsdl` section. Here's what the modified file should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/jax-rpc-ri/xrpcc-config">
  <wsdl name="CorpDataServicesProxy"
        location="http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl"
        packageName="javasoap.book.ch11.glueclient">
  </wsdl>
</configuration>
```

The `name` attribute specifies the model name, just like it did in the RMI example earlier. The `location` attribute contains the address of the WSDL resource for the service. We're going to use the GLUE service from Chapter 9; the WSDL for it is *http://mindstrm.com:8004/glue/urn:CorpDataServices.wsdl*. The `packageName` attribute, like before, specifies the package name of the generated code. This time we'll use `javasoap.book.ch11.glueclient`, since this code is meant to access our GLUE-based service. Generate the client-side code by issuing the command:

```
xrpcc -client -keep -classpath e:\projects\soap -d gen
CorpDataServices.xml
```

This is essentially the same command we used earlier, except that the `-client` option indicates that we're only interested in generating client code. When the process completes, the Java class files should be moved into the proper `javasoap.book.ch11.glueclient` directory so that they appear on the classpath. Take note of the `javasoap.book.ch11.glueclient.Quote` class. This is a custom type that represents the data returned from the GLUE service; it is already defined as a type in the WSDL document. Great! We don't have to do anything special to handle that type.

Let's write a client application called `javasoap.book.ch11.glueclient.StockQuoteApp`. It will look similar to our other `StockQuoteApp`, but it will talk to our existing GLUE-based service. Here's the code:

```
package javasoap.book.ch11.glueclient;

public class StockQuoteApp {
   public static void main(String[] args) {
      try {
         CorpDataServiceImpl proxy =
             new CorpDataServiceImpl(  );
         CorpDataServiceSoap_Stub stub =
             (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap(  );
         stub._setTargetEndpoint(args[0]);
         Quote quote = stub.getQuote("IBM");
         System.out.println("Price is " + quote.get_lastPrice(  ));
         System.out.println("Volume is " + quote.get_volume(  ));
         System.out.println("Timestamp is " + quote.get_timeStamp(  ));
         System.out.println("Change is " + quote.get_change(  ));
      } catch (Exception ex) {
        ex.printStackTrace(  );
      }
   }
}
```

Running this application requires that the URL for the service endpoint be passed on the command line. In this case the proper URL is *http://mindstrm.com:8004/glue/urn:CorpDataServices*. When I ran this I got the following output:

```
Price is 105.09
Volume is 10726400
Timestamp is Mar  8
Change is 1.38
```

We could use the code from this example to make our earlier service talk to the GLUE service to get its data, instead of returning dummy data the way it does now. At this point you should be pretty clear on how to go about making that change, so I'll just run through it quickly. We'll need to modify the `getStockQuote( )` method of the `StockServiceProxy` class. Here's what the modified code looks like:

```
package javasoap.book.ch11.services;

import javasoap.book.ch9.services.ProxyQuote;
import javasoap.book.ch11.glueclient.*;

public class StockServiceProxy {

   public ProxyQuote getStockQuote(String symbol) {

      ProxyQuote pq = new ProxyQuote(  );
      try {
         CorpDataServiceImpl proxy =
             new CorpDataServiceImpl(  );
         CorpDataServiceSoap_Stub stub =
             (CorpDataServiceSoap_Stub)proxy.getCorpDataServiceSoap(  );
         stub._setTargetEndpoint(
             "http://mindstrm.com:8004/glue/urn:CorpDataServices");
         Quote quote = stub.getQuote(symbol);
```

```
            pq.setLast(quote.get_lastPrice(  ));
            pq.setVol(quote.get_volume(  ));
            pq.setTime(quote.get_timeStamp(  ));
            pq.setDiff(quote.get_change(  ));
        }
        catch (Exception ex) {
            ex.printStackTrace(  );
        }

        return pq;
    }
}
```

The `javasoap.book.ch11.glueclient` package is imported; that package contains the stub code for talking to the `urn:CorpDataServices` service. This time we hardcode the URL passed to `_setTargetEndpoint( )`. After getting an instance of `Quote` back from the call to `getQuote( )`, we use it to populate an instance of `ProxyQuote`. Remember, the two services don't return the same data type. Then we simply return the `ProxyQuote` instance. This is pretty painless, thanks to WSDL. Of course, we had to compile the modified `StockServiceProxy` class, and create and deploy an updated WAR file to include the modified code along with the entire `javasoap.book.ch11.glueclient` package.

11.6 Dynamic Invocation Interface

At times, you want more control over the way your application interfaces with a service. One reason for wanting more control is that your application does not know the service methods and parameters in advance, making it impossible to generate stubs. It's also reasonably likely that the service you want to invoke does not provide a WSDL document describing it. The Dynamic Invocation Interface (DII) is an API that allows you to invoke services at a level much closer to the JAX-RPC runtime. We're not going to spend any time with DII, but you should be aware of it. We saw a glimpse of its structure in Chapter 9 when we developed a simple Axis example. It may not be identical, and these APIs are still evolving, but the approach is the same.

I see DII as a last-resort approach to writing code to communicate with SOAP web services. Letting a tool generate the low-level code based on a WSDL description is really the best choice, since WSDL is language and implementation independent, and is an accepted standard. If your SOAP tools can work with a WSDL file, that's your best bet.

11.7 JAXM, in Less Than a Nutshell

The APIs that JAXM provides for writing messaging clients and services are at a low level, similar to what we saw in Chapter 8. They require you to make a connection to the server (or an intermediary) and build up the XML that constitutes the message. Although this is a powerful technique, it gives you a lot of rope with which to hang yourself. I like the flexible nature of working directly with XML, but I still want the structure of RPC-style interaction. If you need to design your services to accept XML documents as parameters, you might find that literal encoding is sufficient within the context of RPC. If that's not enough, then you'll have to walk the messaging (or proprietary) path.

I suggest you take a look at the examples that come with the Java Web Services Developer Pack tutorial. Even though we don't develop a JAXM example here, you're armed with

enough knowledge to understand what's happening. And something tells me that a future edition of this book will delve deep into JAXM, JAX-RPC, and the rest of the JAX Pack.

11.8 What Next?

You're now more than ready to start developing web services in Java. Don't be put off by the pace of change. Keep track of the latest technologies, including new releases of existing implementations. New versions of GLUE, Axis (Apache SOAP), .NET, and many others will make the development of services in Java easier, better, and more powerful. Finally, keep an eye on standardized APIs, security, service registries, and IDE integration; the list is growing as fast as the bandwidth requirements! Jump in.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Java™ and SOAP* is a red firefish, Pterois volitans. These fish are found throughout the warm, tropical waters of the Indian and Pacific Oceans at depths up to 100 feet. They tend to sit nearly motionless on the ocean floor, often under ledges, waiting for potential prey-mainly small fish and crustaceans-to wander into range.

Red firefish are members of the family Scorpaenidae-scorpionfishes-named for the poisonous spines on their dorsal fins. Confident in their ability to defend themselves, firefish often do not back off even when a human approaches; instead, they point their venomous spines towards aggressors. Their confidence is well-placed: the sting of their spines, though not usually fatal to humans, is extremely painful.

In addition to their native ocean habitat, red firefish can also be found living in many home tropical aquariums. They can grow up to 16 inches in length and so require a fairly large tank. But despite their fearsome spines and predatory nature, they're actually very peaceful and sociable, and they get along well with other fish-except the ones small enough to be eaten.

Emily Quill was the production editor and copyeditor for *Java™ and SOAP*. Sarah Sherman and Matt Hutchinson provided quality control. Philip Dangler, David Chu, and Julie Flanagan provided production assistance. Ellen Troutman-Zaig wrote the index.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

Melanie Wang designed the interior layout, based on a series design by David Futato. Neil Walls converted the files from Microsoft Word to FrameMaker 5.5.6 using tools created by Mike Sierra. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Leanne Soylemez.